International Journal on Artificial Intelligence Tools

A Self-Acquiring Knowledge Process for MCTS

André Fabbri*, Frédéric Armetta†, Éric Duchêne** and Salima Hassas‡
Univ de Lyon, Université Lyon 1, LIRIS UMR5205, F-69622, Lyon, France
* *andre.fabbri@liris.cnrs.fr,* † *frederic.armetta@liris.cnrs.fr,*
** *eric.duchene@liris.cnrs.fr,* ‡ *salima.hassas@liris.cnrs.fr*

Received (Day Month Year)
Revised (Day Month Year)
Accepted (Day Month Year)

MCTS (Monte Carlo Tree Search) is a well-known and efficient process to cover and evaluate a large range of states for combinatorial problems. We choose to study MCTS for the Computer Go problem, which is one of the most challenging problem in the field of Artificial Intelligence. For this game, a single combinatorial approach does not always lead to a reliable evaluation of the game states. In order to enhance MCTS ability to tackle such problems, one can benefit from game specific knowledge in order to increase the accuracy of the game state evaluation. Such a knowledge is not easy to acquire. It is the result of a constructivist learning mechanism based on the experience of the player. That is why we explore the idea to endow the MCTS with a process inspired by constructivist learning, to self-acquire knowledge from playing experience. In this paper, we propose a complementary process for MCTS called BHRF (Background History Reply Forest), which allows to memorize efficient patterns in order to promote their use through the MCTS process. Our experimental results lead to promising results and underline how self-acquired data can be useful for MCTS based algorithms.

Keywords: Monte Carlo Tree Search; Computer-Game; Reinforcement Learning; Knowledge Engineering

1. Introduction

In this paper, we propose a self-acquiring knowledge process to deal with the resolution of hard combinatorial problems. The generic MCTS enhancement is applied to a difficult combinatorial game: the game of Go. We observe that MCTS is a very efficient process to cover a huge set of states. Nevertheless it does not take full advantage of its experience. This statement motivates us to explore a new approach to increase the ability for such a process to capitalize and re-use its experience.

The game of Go is a good testbed for Artificial Intelligence [1]. The rules are simple but capturing the underlying explanations for an efficient sequence of moves remains an open problem. The human players acquire an advanced internal representation of the game by an extensive practice. This explains why the best players still defeat computer programs. Indeed a better representation allows to focus on the most relevant parts of the game. Comparatively, a default tree search would instead consider all the possible evolutions of the game. In order to cope with the combinatorial hardness of the problem, a recurrent approach has been to endow the programs with a large amount of encoded expert knowledge

(rules, patterns, etc.).

MCTS led to a major breakthrough for the game of Go[2] and is now applied to a wide range of problems[3]. Contrary to the former approaches, the evaluations of possible evolutions are learned on-line, through random simulations. The program acquires hence some knowledge about the current state by a self-play simulated experience. Nevertheless, MCTS does not suffice to overcome the combinatorial complexity of the game of Go yet. The performance of the programs levels off whatever the additional simulation time allowed and the supplementary expert knowledge provided [4]. In our understanding, after a certain threshold, the pure computational approach cannot be a substitute for a better cognitive integration of the experience.

A promising way to increase the efficiency of a program would be to enhance its ability to accumulate knowledge about its experience simulated. The general idea consists in a better assimilation of the inherent knowledge associated with the states covered by MCTS. This approach has been partially considered in the literature but we claim and argue that this kind of process can be improved in many ways. With our approach BHRF (Background History Reply Forest), we choose to endow the program with the ability to memorize patterns learned on-line and adapt their estimated value during the game. These patterns influence back the simulations in order to enrich the simulated experience. This paper gives insights about the potential of such an approach. Note that our results mainly focus on the quality of the learning rather than on the effective performance in a competitive setting.

More details about BHRF will be provided in Section 3. The MCTS baseline and the main knowledge endowment will be presented in Section 2. Experimental results are given and analyzed in Section 4. A conclusion and some perspectives are drawn in Section 5.

2. How to complement MCTS ?

MCTS progressively weights by self-play several possible evolutions of the game. However, additional knowledge can substantially enhance the learning process. A brief presentation of the MCTS process along with its dynamic is presented in Section 2.1. Section 2.2 reviews the main enhancement in the current programs based on MCTS and Section 2.3 details the underlying data structure.

An extensive presentation of MCTS and its enhancements is beyond the scope of this paper, we invite the reader to refer to a more detailed survey[3].

2.1. *Monte Carlo Tree Search*

The standard MCTS algorithm gradually expands a search tree starting from the current state. The four steps *descent*, *growth*, *roll-out* and *update* (see Figure 1) are iteratively applied until we meet some restraining constraints (time, memory or iteration number). The *descent* policy covers the tree and selects a new node to sample. The *growth* phase adds a node to the tree search. From this node, the *roll-out* policy generates the remaining moves until the simulation reaches a final state. The *update* phase finally propagates back the outcome in the tree search.

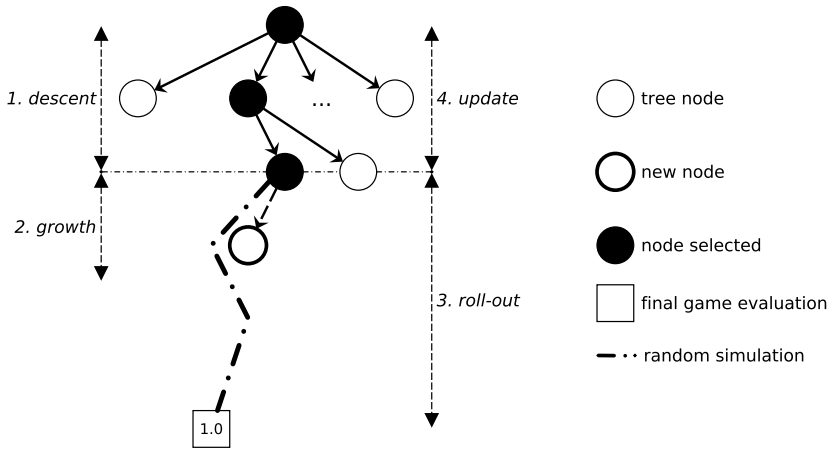


Figure 1. Monte Carlo Tree Search process

The values learned in the search tree are tightened with the underlying policies. On the one hand, the *descent* policy considers the node's values to reach the most promising one to deepen. On the other hand, the outcome of the simulated end game adjusts the values of the node selected during the *descent* and influences back the next *descent* policy. The interaction between the policy and the learned values refers to the *generalized policy iteration* process[5].

In MCTS, the policy iteration involves also a *roll-out* policy. This policy generates the last moves leading to the final state and therefore contributes to the learning process. However the *roll-out* policy does not benefit from the learned weights. The purpose of the proposed method is to influence back the *roll-out* policy as presented in Figure 2.

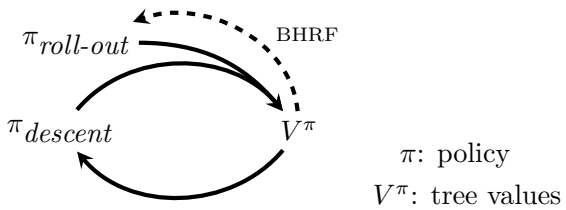


Figure 2. Generalized policy iteration process for MCTS

2.2. *Enhanced policy iteration process*

As pointed out by the generalized policy iteration, the policies play a major role in the learning. The *descent* and *roll-out* policies have been progressively enhanced to cope with

the issues addressed by each phase. In this section, the main enhancements for each policy are reviewed from a Go-specific and a more general perspective.

Over the iterations, the node's weights are progressively refined and the *descent* policy has to focus quickly on the most promising parts of the tree. For the game of Go, expert off-line knowledge may efficiently promote states subsequent to interesting moves and avoid silly ones. This knowledge may enhance the search by biasing the values or pruning the tree[6]. From a broader perspective, the Upper Confidence bound applied to Trees [7] considers the number of updates to achieve a good balance between the exploration of current sub-optimal states and the exploitation of the current best states.

A pure random *roll-out* policy generates many non-representative final states whose outcome slows the learning of the system. Thus the *roll-out* policies generally involve additional knowledge to enhance the relevance of the final states. For the game of Go, the sequence-like policies successfully consider expert or off-line knowledge to guide the simulations[8]. Such a *roll-out* policy is difficult to improve because it has to balance carefully the distribution of the final states to cover [9]. A promising way consists in designing adaptive *roll-out* policies rather than static ones.

As presented in the next section, the expert knowledge involved in sequence-like policies is not suitable for an adaptive policy. General-game purpose methods such as N-Grams[10], Last-Good-Reply[11] or an application of Win/Loss states[12] propose a more adaptive knowledge representation. Such methods enhance the *roll-out* policy with small patterns (spatial or temporal) evaluated on-line. However the patterns considered arise from the *roll-out* itself rather than the search tree.

Previous attempts to exploit knowledge coming from the tree have been mostly limited to single moves[13,14,15a]. To our best knowledge, the Pool-RAVE enhancement[13] is the best attempt applied to the game of Go so far. A pool of potential best moves are picked up during the *descent* and re-exploited in the *roll-out*. This method achieves good results for the *roll-out* policies without expert knowledge but does not intend to learn explicit knowledge from the tree. Such a learning requires the adequate underlying data-structure as presented in Section 2.3.

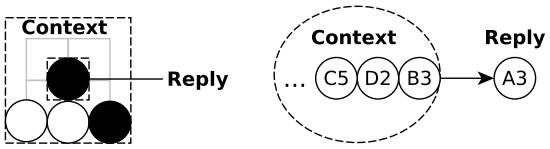
The search tree actually stores the outcomes of the simulations. Following this perspective, MCTS becomes then a cognitive problem: how to capitalize the simulated experience of the system? This is a long-term issue and, in our approach, we will focus on the memorization of raw moves sequences coming from the tree.

2.3. *Overall view of additional data structures*

The policies select the action to perform based on the knowledge available for the system. The data-structure supporting this knowledge has a high influence on how this knowledge may be re-exploited. Current programs based on MCTS handle different kinds of knowledge. In the present paper, we differentiate the knowledge learned in the search tree from the additional knowledge considered in the enhanced policies. The latter can be applied

[a]The only exception is the second-order history heuristic[15] which considers also the previous one

to different situations contrary to the node's knowledge which is specific to a single game state. In this section, the existing complementary data-structures for MCTS are reviewed for both kinds of knowledge.



(a) Spatial pattern (b) Temporal pattern

Figure 3. Additional knowledge for MCTS

In the best Go-program, the expert knowledge involved in *descent* and *roll-out* policies considers immediate reply to local spatial contexts. For each pattern, the surrounding positions stand for the context and the middle move corresponds to an appropriated reply (Figure 3a). This knowledge successfully simulates local fights in sequence-like policies but is not prone to any adaptive policy. For the same spatial context, only one reply is possible: the one in the middle. Therefore the evaluation of such a pattern is not related to the relevance of the last reply but to any move inside the whole pattern[16] (whether the context or the reply). To propose a truthful comparison among all the possible replies, the pattern could share the same context as in temporal patterns.

The general-game approach considers mainly temporal patterns, i.e, the immediate reply to a sequence of immediately previous moves. For each pattern, the first moves stand for the context and the last move stands for an appropriated reply (Figure 3b). This context is generally small. N-grams [10] and LGR methods[11] consider up to two moves for the context. TO-MAST[14] and pool-RAVE [13] actually consider only the reply move itself. However this short term perimeter for the contexts may raise a short-sighted phenomenon, i.e., the so formed sequences can be applied to many different states but are not relevant for each of them. Previous attempts such as the move answer tree[17] or the local tree[18] propose to specialize the temporal pattern but does not provide effective results yet. Spatial and temporal patterns have been already combined [12] for the game of Go. The temporal sequence considered involves a spatial pattern rather than single moves but their size has been limited to a context of size one (an immediate reply pattern to the previous one).

The values learned in the search tree corresponds to the estimated win probability of the specific game states covered during the *descent* phase. This knowledge is not prone to be re-exploited in similar states (except for the very same game state see [19]). Indeed, if a branch of the tree learns a sequence of actions that solves a local sub-problem, this sequence has to be rediscovered in the other branches where this sub-problem occurs [20]. Moreover each time the opponent has played his move, one has to prune the tree to keep only the subtree associated with the moves that are effectively played. As a result, the knowledge accumulated in the other branches is also lost.

The purpose of our approach is to design temporal patterns in order to extract the knowledge inside the search tree. We choose to extend the size of the patterns so that they can specialize to more specific contexts, as for the search tree.

3. A transversal data structure to complete MCTS

Our proposal is generic and can be applied to MCTS whatever the considered problem. The purpose of our approach is to increase the system ability to memorize data and also its ability to adapt it to close applicative contexts. The overall idea of BHRF is presented in Section 3.1 and an implementation is detailed in Section 3.2.

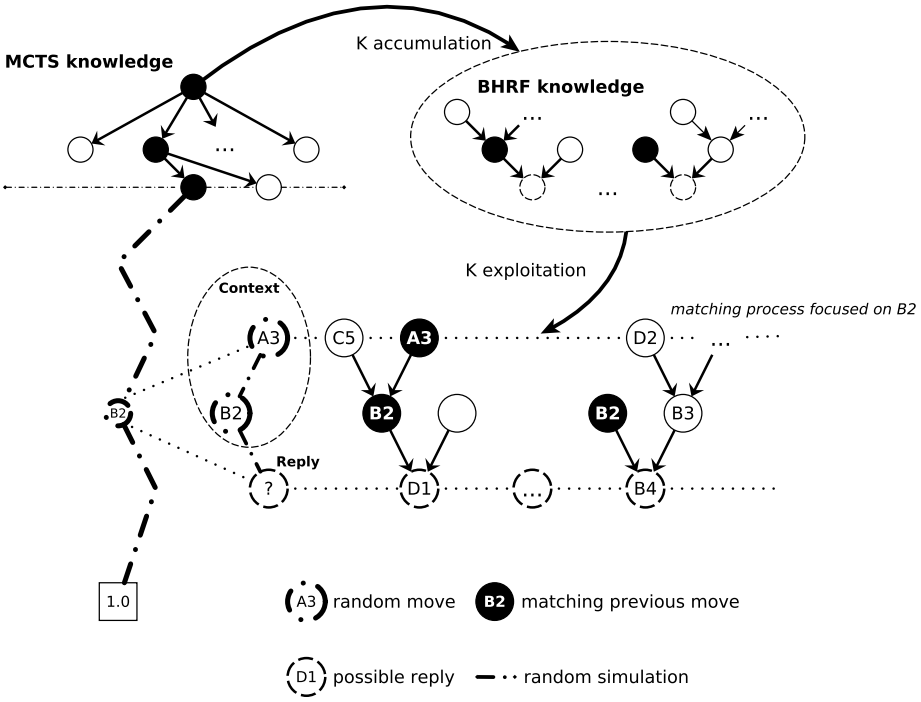


Figure 4. Background History Reply Forest process

3.1. *Background History Reply Forest*

We propose to build up an independent data-structure to complement the Monte Carlo tree and allow transversal knowledge memorizing. In order to improve the *roll-out* reliability, we choose to exploit the self-acquired and long-term data to influence the *roll-out* policy. Moreover the purpose of this contribution is to produce a policy that improves the learning process using knowledge that is already gathered in the search tree. That is why, in our

proposal, the complementary data structure is updated according to the current sequence played during the *descent* phase and the simulation outcome (Winning, Loosing).

The complementary data structure presented here accumulates small temporal patterns of increasing size, thanks to a forest data structure. The root of each tree stands for a reply while considering a context formed by a path between a leaf and the root. The set of all reply trees defines the Background History Reply Forest (BHRF). As the search tree of MCTS progressively expands, the reply trees of BHRF grows concurrently. Though the new nodes added to BHRF trees form to new temporal patterns. Each new pattern has its context extended by one more preceding action with respect to a pattern already gathered by BHRF. As presented in Figure 4, the temporal patterns accumulated are then considered by the *roll-out* policy.

Similar methods involving temporal pattern[11,10] learn the temporal patterns according to the moves generated during the *roll-out*. However the *roll-out* policies focus mainly on the position around the last move generated. The originality of this method is to learn temporal pattern from the search tree, i.e., according to the moves selected during the *descent* phase. As a consequence, the evaluation of temporal patterns consider the exact configuration associated with the search tree.

The moves played during the *descent* phase are selected and evaluated according to the whole board configuration. Therefore the tree search is more able to handle complex configurations arising during the game such as Ko fights. Hence BHRF benefits to a certain extent from the global understanding of the search tree and can consider complex moves such as Ko threats; provided that these moves have been sufficiently explored curing the *descent* phase [21].

3.2. *Knowledge accumulation and exploitation*

During the *update* phase, MCTS and BHRF are independently updated using the same simulation outcomes. The knowledge acquisition of BHRF is detailed through two algorithms. We cover each move of the *descent* sequence and launch a reply tree update (Algorithm 1). Indeed, the whole *descent* sequence contains many different context-reply associations to memorize. The details of the tree update process is presented in Algorithm 2. As for MCTS, the node estimates are updated each time a stored sequence matches with the current one. For each reply tree, new nodes are regularly added and the whole structure is kept over the turns. Unlike Last Good Reply[11] or N-grams[10] approaches, we do not restrain the maximal size of the context (the size of patterns is bounded by the depth of the search tree). The number of temporal patterns added after each simulation is then bounded by the size of the *descent* sequence.

During the *roll-out* phase, the knowledge accumulated in BHRF influences back the *roll-out* policy. The ϵ parameter sets a priori the exploitation rate of BHRF during the simulation process. If BHRF is not applied or BHRF does not suggest any reply, the default policy is applied. The knowledge exploitation of BHRF applies temporal patterns according to their contexts and their estimated quality.

As a first step, the policy considers only the temporal patterns whose the reply complies

Algorithm 1 *update* algorithm - Reply Forest

```
procedure UPDATEREPLYFOREST(descentSeq: Array<Move>, outcome: Result)
//descentSeq: moves selected in the last tree descent
//outcome: result of the simulation following the descent i.e {Win, Loss}
    for i ← descentSequence.size −1 to 0 do
        rt: ReplyTree
        rt ← self.getReplyTree(descentSeq[i])
// the reply tree leading to move i is considered
        if opponentMove(i) then
            rt.updateTree(i,descentSeq,inv(outcome))
        else
            rt.updateTree(i,descentSeq,outcome)
        end if
    end for
end procedure
```

Algorithm 2 *update* algorithm - Reply Tree

```
procedure UPDATETREE(i: int, descentSeq: Array<Move>, outcome: Result)
//i: position of the reply move in the sequence descentSeq
    nodeCreated: Boolean
    mv: Move
    nodeCreated ← false
    childNode, lastNode: Node
    lastNode ← self.getRoot()
      // the root node of the tree corresponds to the reply
    while i > 0 && ¬ nodeCreated do
        i ← i −1
        mv ← descentSeq[i]
        childNode ← lastNode.getDirectChild(mv)
        if childMove == null then
            childMove ← lastNode.createChild(mv)
            childMove.updateMean(outcome)
            nodeCreated ← true
        else
            childMove.updateMean(outcome)
            lastNode ← childNode
        end if
    end while
end procedure
```

with the rules and the context matches exactly the last moves generated. Since a richer context defines a more accurate knowledge, we keep only the more accurate patterns with the highest number of compatible previous moves. Hence all the remaining patterns share the same matching context length. As a second step, the appropriate reply is selected among the remaining patterns according to its evaluation. We propose here two different selection processes: `soft-UCT` and `last-reply`.

The `soft-UCT` process computes an estimate for each remaining pattern (Algo-

rithm 3.). The estimate is inspired from the initial UCT formula[7]. Though the UCT algorithm selects deterministically the next action according to the complete description of the board state, as for the nodes in the search tree. In BHRF, we have only a partial description of the board state. Therefore the reply is selected according to the UCT formula in a softmax version as follows[b]:

$$P(r|c) = \frac{\bar{x}_{r|c} + b \times \sqrt{\frac{\ln \sum_{i \in \mathscr{C}} n_{i|c}}{n_{r|c}}}}{\sum_{i \in \mathscr{C}} P(i|c)}, \tag{1}$$

where

r	: legal reply	$\bar{x}_{r\|c}$	: average result of r in c
c	: context	b	: UCT bias term
$\mathscr{C}$	: set of legal replies for context c	$n_{r\|c}$	: selection number of r in c

Since only a few BHRF patterns generally match the same context, the selected reply is eventually played according to its UCT estimate, as presented in Algorithm 3. The estimate is computed each time we use BHRF during the *roll-out* phase. Hence this selection process is more accurate, at the expense of a computational overhead.

Algorithm 3 *roll-out* policy with `soft-UCT` selection process

```
procedure ROLLOUTPOLICY(lastMoves: Array<Move>) : Move
//lastMoves: moves previously selected (temporal context)
    candidate: Move
    lstCandidate: List<Move>
//Probability ϵ to use BHRF
    if randomValue() < ϵ then
      // Get the legal replies associated to the longest matching patterns
        lstCandidate ← getReplyForest().longestMatchingPattern(lastMoves)
        lstCandidate ← getReplyForest().checkLegalReplies(lstCandidate)
      // Select and apply the reply according to the UCT estimate (Equation 1)
        candidate ← softMaxUctSelection(lstCandidate)
        if randomValue() < candidate.uctEstimate() then
            return candidate
        end if
    end if
// Plays default policy otherwise
    return defaultPolicyMove()
end procedure
```

However, the temporal pattern of BHRF are progressively accumulated and updated according to the actions selected during the *descent* phase. Following this perspective, one can note that the selection process of BHRF mimics a selection process occurring already

[b]the bias term b has been set to 0.7 empirically

during the *descent* phase. Indeed, each action selected during a *descent* phase may be seen as the reply in response to the moves that have been previously selected during this phase.

The `last-reply` process simplifies the selection process of BHRF by considering directly the decisions that have already been adopted during a previous *descent* phase (Algorithm 4). Among the longest matching patterns, the appropriate reply is the last that has been selected in a previous *descent* phase, for this context. This selection process is inspired by the Last Good Reply enhancement[11]. However in this case, the replies arise from the accurate selection that occurs during the *descent* phase, rather than from the random sampling of the *roll-out* phase. Due to computational costs, we do not check for a legal reply during the selection process. If the reply suggested by BHRF is not legal, the default policy is applied.

Algorithm 4 *roll-out* policy with `last-reply` selection process

```
procedure ROLLOUTPOLICY(lastMoves: Array<Move>) : Move
//lastMoves: moves previously selected (temporal context)
    candidate: Move
    lstCandidate: List<Move>
//Probability ϵ to use BHRF
    if randomValue() < ϵ then
// Get the replies associated to the longest matching patterns
        lstCandidate ← getReplyForest().longestMatchingPattern(lastMoves)
// Select the last updated pattern and play it if legal
        candidate ← lastReplySelection(lstCandidate)
        if legalMove(candidate) then
            return candidate
        end if
    end if
// Plays default policy otherwise
    return defaultPolicyMove()
end procedure
```

4. Experimental results

In this section, we study the influence of BHRF knowledge over the learning process. Unlike the expert patterns involved in competitive programs, this knowledge does not rely on prior considerations. As a consequence, BHRF patterns are self-acquired by the program along with MCTS, according to its own simulated experience.

The experimental protocol is detailed in Section 4.1. The results presented in Section 4.2 and Section 4.3 show that BHRF successfully catches the knowledge of the tree search. In Section 4.4 we highlight that this tree knowledge may successfully complement an expert *roll-out* policy and Section 4.5 compare the results obtained according to the different selection processes proposed. Finally, we draw in Section 4.6 a first understanding of the BHRF approach according to a cognitive perspective of Artificial Intelligence.

4.1. *Experimental setup*

The BHRF heuristic has been implemented using the open source framework *Fuego* (version 1.1)[22]. This framework offers the main enhancements for MCTS computer-go programs such as UCT and expert knowledge. In this program, the expert knowledge is used to initialize the new node of the tree search and also for the *roll-out* policy.

In the following experiments, the program with the BHRF heuristic competes against the same baseline program without BHRF (All game results are provided with 95% confidence interval). The common settings of both programs are the same (if not mentioned). The settings we will further consider in the experiments are the following:

- Board size (△): `9x9,19x19`: determines the difficulty of the game played. The search space is huge on 19x19 and the program has to focus even more on game state of interest. Moreover games on wider boards produce more complex situations which may not be covered by expert knowledge.
- *roll-out* simulations (▼): `1k, ..., 10k, 30k`: corresponds to the maximum number of simulations granted. A larger value generates a more accurate tree knowledge and therefore a better *descent* policy.
- *roll-out* expert knowledge (■): `True, False`: defines whether the *roll-out* policy involves expert knowledge or not.
- BHRF: ϵ `= 0 .. 100%` (□): tunes the rate of exploitation of the self-acquired knowledge in the *roll-out* phase.
- Selection process: `soft-UCT` or `last-reply` (⧫): defines how to select the appropriate reply between different patterns sharing the same context. Most results are presented for the `soft-UCT` selection process.

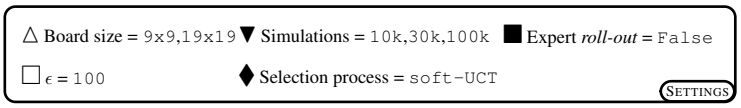
In this article we mainly focus on the potential for using such a knowledge, rather than on the next-step optimisation. That is why we consider both programs with an equal number of simulations rather than an equal computing time. Considering our lightly optimized BHRF algorithms and a middle range hardware configuration (Intel(R) Core(TM) i7-2600 CPU 3.40 GHz with 8GB memory), the BHRF module tends to slow down the computing time from 4 to 12 times with the `soft-UCT` selection process, according to the game size and the *roll-out* simulation number. The current implementation is not competitive on equal time settings but provides a substantial improvement in the learning quality.

4.2. *Increasing efficiency due to self-acquired knowledge*

For these experiments, competing programs are both set without expert *roll-out* knowledge but both programs initialize the node value using prior expert knowledge. In Table 1, we report the values of BHRF knowledge for 9x9 and 19x19 game sizes with the maximum number of *roll-out* simulations allowed.

The program that considers self-acquired knowledge, significantly outperforms the baseline program in all the configurations and whatever is the number of allowed simulations. Pool-RAVE[13] provides similar results for the game of Go without expert *roll-out* knowledge (their results are provided only for the 9x9 game size). These results confirm

Table 1. Success rate for the BHRF approach against a random *roll-out* policy



Simulations	**10000**	**30000**	**100000**
Goban 9x9	$+17.3\% \pm 1.6$	$+16.9\% \pm 2$	$+18\% \pm 2.6$
Goban 19x19	$+24.3\% \pm 3.5$	$+26.6\% \pm 2.5$	$+27.7\% \pm 3.4$

further the interest of using knowledge from the search tree in the *roll-out*.

The BHRF usage may explain the difference in the results between both sizes. The game length on a 19x19 goban are bigger than the games on a 9x9 goban. In the same way, the *roll-out* are longer on a 19x19 goban and, as a consequence, BHRF has been more exploited for this size. Moreover 19x19 game size has a huge combinatorial space. A more careful move generation has more influence on the resulting performances. This may explain the slight improvement on 19x19 as the number of simulations increases.

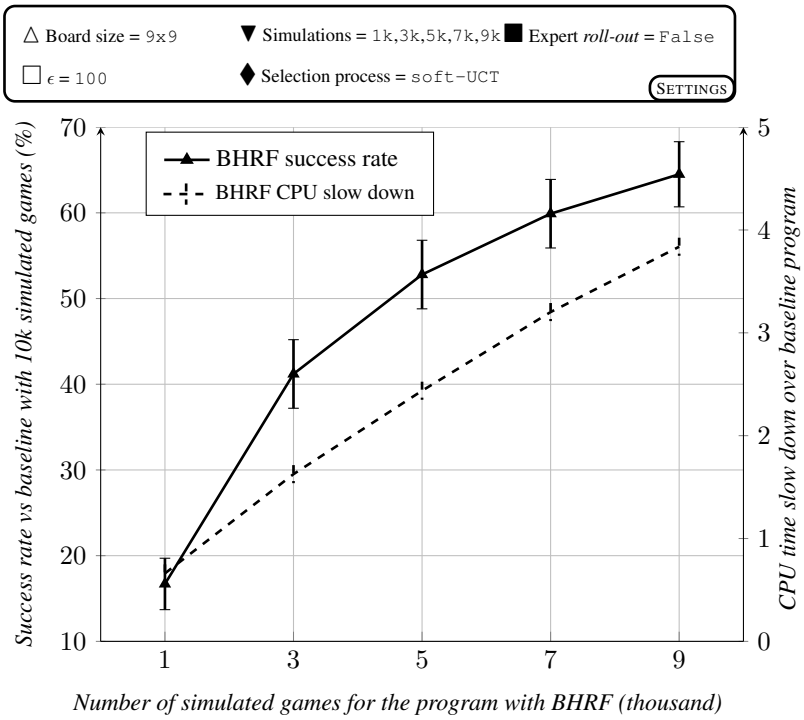


Figure 5. Success rate and CPU time ratio for BHRF approach considering *roll-out* number variations (opposed to the same configuration without BHRF, *roll-out* number fixed to *10k*)

In order to appreciate the BHRF ability to manage and benefit from complementary knowledge, we choose to vary the available number of *roll-out* simulations available for

BHRF, while keeping it constant (fixed to *10k*) for the baseline program. As shown in Figure 5, BHRF outperforms the baseline program as soon as it reaches the half of the available number of *roll-out* simulations of the baseline program. However, as mentioned previously, the process of real-time self-acquiring knowledge is time-consuming and a further optimization is required before being time-competitive with the current programs.

4.3. *Quality of the self-acquired knowledge*

The purpose of our proposal is to accumulate the experience occurring through the general MCTS process in order to consider it during the *roll-out* phase (see Figure 4). In this section, our main objective is to get a better understanding of the way knowledge is acquired by our approach (BHRF).

One can note that the more realistic the *roll-out* moves are, the more relevant is the evaluation for a game state. An unavoidable strategy for Go is to play locally if possible, i.e. , not leaving a local position unstable as far as possible. The sequence-like policies are meant to reproduce this behavior in the professional-level programs such as *Fuego*. These policies consider the positions around the last move played in order to promote local replies. Since all local moves are not relevant, an appropriate local reply is selected according to some tactical knowledge such as the spatial pattern presented in Section 2.3. In this section, we assess the *roll-out* moves in the light of such a behavior. We define a *roll-out* profile by considering the two criteria presented on Figure 6. These criteria are:

- The ability to play locally: a move is local if it is played within a Manhattan distance of three from the previous one (Section 4.3.1).
- The ability to play local patterns: a move corresponds to a local pattern if the move is local and matches one of the spatial pattern considered by sequence-like policies (Section 4.3.2).

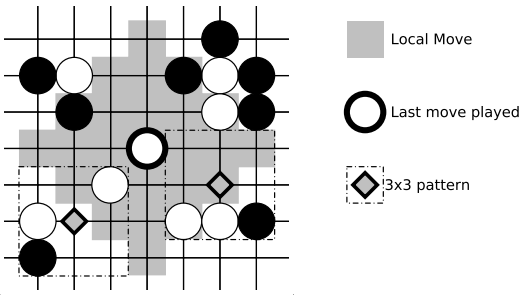


Figure 6. Criteria of the *roll-out* profiles

In particular, *Fuego* considers spatial patterns and local play in the *roll-out* and the *descent* policies: in the *descent* policy to tune the initial values and, in the *roll-out* policy, to generate the moves in a sequence-like manner. This expert *roll-out* policy is defined *a-priori* according to explicit rules. Hence we investigate to what extent BHRF reproduces

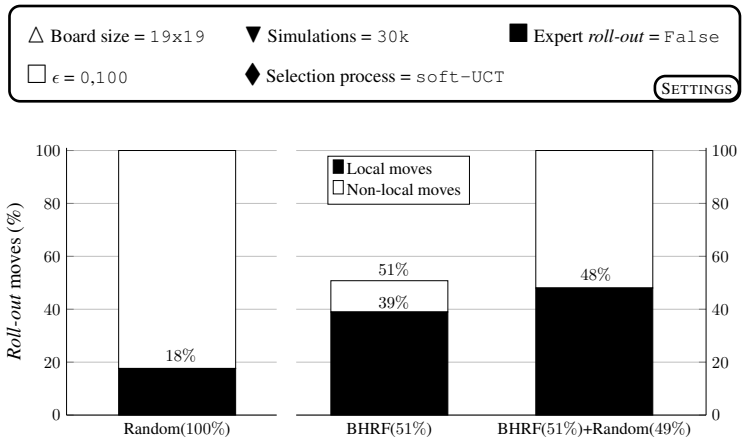
such a sequence-like behavior in the *roll-out*, without explicit prior knowledge about it. We compare the profile of BHRF moves with two other profiles:

- A random *roll-out* profile, obtained with *Fuego* set without expert *roll-out*.
- An expert *roll-out* profile, obtained with default *Fuego* (with expert *roll-out*).

In order to obtain the profile of BHRF moves only, we disable the *Fuego* expert *roll-out* policy in our program. The propensity to use BHRF is set to its maximum (ϵ equals 100%). Hence, each time the current sequence matches a context memorized by BHRF, the selected reply is played according to its UCT estimate (see Algorithm 3). BHRF feeds about 51% of the *roll-out* moves and the other moves are played at random.

4.3.1. *Ability to play locally*

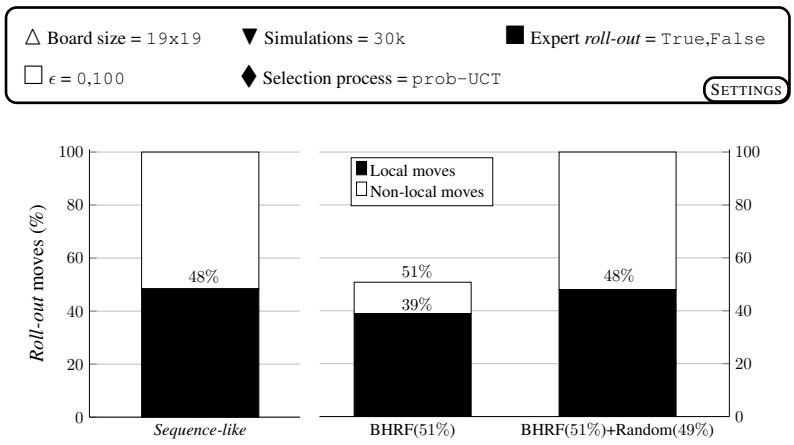
In Figure 7, we compare the profiles of *roll-outs* resulting from a raw baseline program (no expert *roll-out* data) with or without the use of BHRF. One can note that BHRF tends to show more locality than the random *roll-out* strategy. To our point of view, this local play explains in a large part why BHRF defeats a random *roll-out* policy (cf. Section 4.2).



(a) Random policy ($\epsilon = 0$) (b) Random policy with BHRF ($\epsilon = 100$)

Figure 7. *Roll-out* profiles: moves locality (Random approach vs BHRF approach)

As we can see from Figure 8, the BHRF approach allows to play as much locally as the expert approach (48%), though both policies are not equivalent. Indeed, local play in expert *roll-out* policies is defined *a-priori* according to explicit rules whereas BHRF retrieves it from the search tree. However all the local moves are not relevant and, as mentioned earlier, the spatial patterns considered in sequence-like policies are a way to identify the most promising ones.

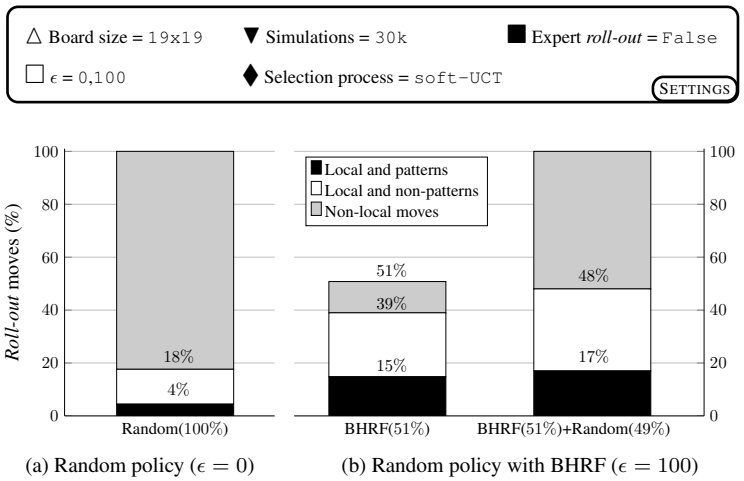


(a) Expert policy (b) Random policy with BHRF ($\epsilon = 100$)

Figure 8. *Roll-out* profiles: moves locality (*Fuego* approach vs BHRF approach)

4.3.2. *Ability to play local patterns*

In this section, we focus on the ability to play *roll-out* patterns for the selected approaches (pure random, *Fuego* and BHRF). Figure 9 states that the BHRF *roll-out* plays more than four times more local patterns than the pure random one. One can understand that BHRF can acquire relevant patterns that helps the general MCTS process to understand the necessity to play locally. Without these self-acquired patterns, a pure random approach misses the pattern opportunities which considerably decreases the program competitiveness.

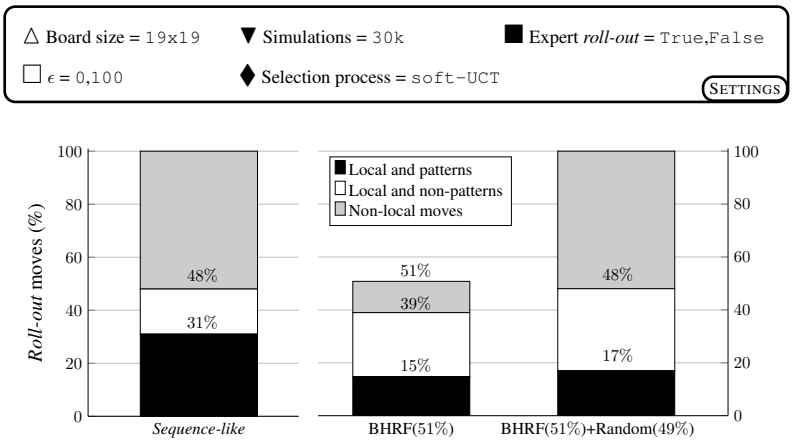


(a) Random policy ($\epsilon = 0$) (b) Random policy with BHRF ($\epsilon = 100$)

Figure 9. *Roll-out* profiles: local move pattern (Random approach vs BHRF approach)

In Figure 10, one can note that expert *Fuego roll-out* plays even more local patterns. In fact, *Fuego* plays local patterns in a more systematic way, and cannot consider the rele-

vance of a pattern move. From our point of view, this is the counterpart of myopic *a-priori* patterns. BHRF copes with locality but can also perceive and memorize patterns adapted to the current situation.



(a) Expert policy (b) Random policy with BHRF ($\epsilon = 100$)

Figure 10. *Roll-out* profiles: local move pattern (*Fuego* approach vs BHRF approach)

4.4. *Combining BHRF with expert knowledge*

In the previous section, we have shown that BHRF produces local moves which do not follow expert spatial patterns in a systematic way. However the temporal patterns memorized by BHRF may, even though, complement sequence-like policies. In this section, we choose to involve expert knowledge heuristics for BHRF *roll-out* as a second choice. When no move is selected by BHRF, the standard rules originating from *Fuego roll-out* policy are applied. BHRF competes then with *Fuego* set to the best of its ability.

In Figure 11, we show that BHRF with a `soft-UCT` selection process outperforms *Fuego* by 11% when we use BHRF data moderately ($\epsilon = 15$). The number of simulation was set to 30k in order to accumulate substantial data about the game. A low ϵ value involves more exploration through the general MCTS process. BHRF nevertheless allows to significantly increase the global performance while memorizing efficient situated patterns.

The expert knowledge involved in the *roll-out* policy plays locally, around the last move generated. On a 9x9 board, local fights quickly cover the whole board, but on a 19x19 board, local fights have also to consider the situation in other areas of the board. As mentioned in Section 4.3, our data-structure embeds both the expert and the self-acquired knowledge of the tree. BHRF knowledge may adjust this locality according to the state covered in the tree. Therefore, the enhanced *roll-out* policy benefits from knowledge adapted to the real situation.

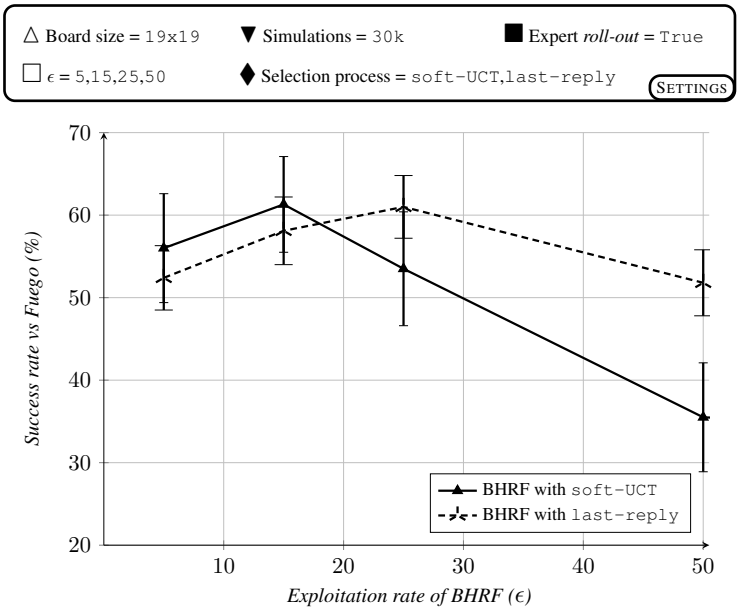


Figure 11. Success rate on 19x19 with an expert *roll-out* policy

4.5. *Comparison between both selection processes*

As presented in section 3.2, we propose two different selection processes for BHRF. In the previous experiments we have consider only the `soft−UCT` process. This process selects the appropriate reply according to the evaluation of the BHRF patterns, which seems more accurate but suffers from a high computational cost. The `last−reply` applies just the last reply that has been previously selected in the *descent* phase for this context. In other words, this selection process suggests to consider, besides the temporal patterns, the decision made in the search tree.

In Figure 11, we plot also the success rates of the `last−reply` selection process with the same settings stated in Section 4.4 (30k simulation and expert *roll-out*). The `last−reply` selection process obtains results similar to the `soft−UCT` selection process: an increase of $+11\%$ success rate. This result is obtained for an higher exploitation of BHRF ($\epsilon = 25$) in this case. Though we remind that `last−reply` does not check during the selection process whether the replies are legal. As a consequence, many replies suggested by BHRF have been probably discarded. This can explain why this result is obtained .

The `last−reply` selection process is faster than the `soft−UCT` process. According to these settings, the computational consumption of the program with `last−reply` is only increased by 2 times compared with *Fuego*. The program with `soft−UCT` requires much more CPU time with these settings (about 12 times). Hence the exploitation of the decisions made during the *descent* phase appears as a promising way to fasten the exploitation process. Though a slow down of 2 times is still consequent with an equal time setting

and would require further optimizations.

4.6. *Toward an autonomous system*

Professional level programs like *Fuego* involve expert knowledge in both policies: in the *descent* policy to bootstrap the values and in the *roll-out* policy as it is. This knowledge is defined *a priori* with a limited number of explicit rules. It is applied equally for every board position and every game possible. The *Fuego roll-out* policy does not benefit from any feedback, whereas the results of the simulations may balance the bias introduced by this prior in the tree. Therefore the expert knowledge considered have to suit to the largest number of possible positions.

As presented in Figure 2, BHRF aims to provide an adaptive *roll-out* policy based on knowledge coming from the search tree. In Section 4.3, we have shown that BHRF successfully embeds the prior expert knowledge in the tree. It appears difficult to bypass this kind of long-term expert knowledge when it is available. On the other hand, the result presented in Section 4.4 suggests that BHRF learns new patterns which ones successfully complement the *Fuego roll-out* policy. Moreover we have shown in Section 4.5 that the decisions themselves made during previous *descent* phases can be successfully applied during the *roll-out* through BHRF.

From a cognitive perspective, human players gain expertise through the acquisition of perceptual patterns. The progressive enhancement of such patterns may explain the difference of skills between a professional and an amateur player[23]. Following this perspective, BHRF applies the following two processes: adaptation of existing pattern and accumulation of new patterns. In order to go further in this way, other processes are necessary to create higher level patterns. For instance, BHRF does not allow generalizations between the accumulated patterns. Indeed the temporal pattern are applied only with an exact context matching, as this is usually done for the game of Go[11,10]. As an example, a fuzzy match may generalize the patterns to much more situations. Moreover the temporal patterns presented here cannot abstract spatial notions such as symmetries or rotational invariance.

5. Conclusion

This paper proposes new enhancements to complete the well-known MCTS search process in the context of combinatorial games. We show that a better assimilation of the knowledge learned by MCTS may enhance the performance of the system. As presented in this paper, the knowledge stored in the search tree is not prone to be re-exploited. A promising way is to consider MCTS as a cognitive system (Section 4.6). Indeed, a better assimilation of this knowledge allows to adapt it to different situations and may avoid to learn redundant patterns among the branches[20] for instance. Moreover, such an approach may provide better insights on how the system considers its simulated experience and therefore the underlying mechanisms of MCTS.

The data-structure detailed in this paper is a raw manner to memorize adaptive knowledge coming from the search tree. The presented results show that this data-structure successfully catches such a knowledge (Section 4.2 and Section 4.3) and this knowledge may

actually complement expert knowledge (Section 4.4). In particular, a professional program combined with BHRF achieves up to a 11% increase in performance. These results points out the potential of such an approach though the slow down of the learning process prevents from experiments with constant time. Following this perspective, we propose a simplified selection process that reaches similar results with a limited amount of additional time (Section 4.5).

We decided to apply our algorithm to the game of Go because this problem is demanding in terms of knowledge, nevertheless the current implementation is designed for a general-game perspective. A more time-efficient implementation may consider characteristics of the game such as the locality of the reply but this was beyond the scope of the present paper.

Acknowledgments

We gratefully acknowledge support from the CNRS/IN2P3 Computing Center (Lyon/Villeurbanne - France), for providing part of the computing resources needed for this work.

Bibliography

1. B. Bouzy and T. Cazenave, Computer Go: An AI oriented survey, *Artificial Intelligence* **132**(1) (2001) 39–103.
2. S. Gelly and D. Silver, Achieving Master Level Play in 9 x 9 Computer Go., in *AAAI* **8**, (Palo Alto, USA, 2008), pp. 1537–1540.
3. C. Browne, E. Powley, D. Whitehouse, S. Lucas, P. I. Cowling, P. Rohlfshagen, S. Tavener, D. Perez, S. Samothrakis and S. Colton, A Survey of Monte Carlo Tree Search Methods, *Computational Intelligence and AI in Games, IEEE Transactions on* **4**(1) (2012) 1–43.
4. A. Bourki, G. Chaslot, M. Coulm, V. Danjean, H. Doghmen, J.-B. Hoock, T. Hrault, A. Rimmel, F. Teytaud, O. Teytaud, P. Vayssire and Z. Yu, Scalability and Parallelization of Monte-Carlo Tree Search, in *Computers and Games*, eds. H. van den Herik, H. Iida and A. Plaat, *Lecture Notes in Computer Science* **6515** (Springer Berlin Heidelberg, 2011) pp. 48–58.
5. R. S. Sutton and A. G. Barto, *Reinforcement Learning: An Introduction* (Cambridge Univ Press, 1998).
6. G. Chaslot, C. Fiter, J.-B. Hoock, A. Rimmel and O. Teytaud, Adding expert knowledge and exploration in monte-carlo tree search, in *Advances in Computer Games*, eds. H. van den Herik and P. Spronck, *Lecture Notes in Computer Science* **6048** (Springer Berlin Heidelberg, 2010) pp. 1–13.
7. L. Kocsis and C. Szepesvári, Bandit Based Monte-Carlo Planning, in *Machine Learning: ECML 2006*, eds. J. Frnkranz, T. Scheffer and M. Spiliopoulou, *Lecture Notes in Computer Science* **4212** (Springer Berlin Heidelberg, 2006) pp. 282–293.
8. Y. Wang and S. Gelly, Modifications of UCT and sequence-like simulations for Monte-Carlo Go, in *IEEE Symposium on Computational Intelligence and Games, 2007.* (Honolulu, HI, 2007), pp. 175–182.
9. D. Silver and G. Tesauro, Monte-Carlo Simulation Balancing, in *Proceedings of the 26th International Conference on Machine Learning* (Omnipress, Montreal, CA, 2009), pp. 945–952.
10. M. J. Tak, M. H. Winands and Y. Bjornsson, N-Grams and the Last-Good-Reply Policy Applied in General Game Playing, *Computational Intelligence and AI in Games, IEEE Transactions on* **4**(2) (2012) 73–83.

11. H. Baier and P. Drake, The Power of Forgetting: Improving the Last-Good-Reply Policy in Monte Carlo Go, *Computational Intelligence and AI in Games, IEEE Transactions on* **2**(4) (2010) 303–309.
12. J. Basaldúa, S. Stewart, J. M. Moreno-Vega and P. Drake, Two Online Learning Playout Policies in Monte Carlo Go: An Application of Win/Loss States, *IEEE Transactions on Computational Intelligence and AI in Games* **6:1** (2014) 1–9.
13. A. Rimmel, F. Teytaud and O. Teytaud, Biasing Monte-Carlo Simulations through RAVE Values, in *Computers and Games*, eds. H. van den Herik, H. Iida and A. Plaat, *Lecture Notes in Computer Science* **6515** (Springer Berlin Heidelberg, 2011) pp. 59–68.
14. H. Finnsson and Y. Björnsson, Learning Simulation Control in General Game-Playing Agents., in *AAAI* **10**, (Palo Alto, USA, 2010), pp. 954–959.
15. P. Drake and S. Uurtamo, Heuristics in Monte Carlo Go., in *IC-AI* (Las Vegas, USA, 2007), pp. 171–175.
16. Ł. Lew, *Modeling Go Game as a Large Decomposable Decision Process*, PhD thesis, Warsaw University - Faculty of Mathematics, Informatics and Mechanics, (Warsaw, UA, 2011).
17. H. Baier, Adaptive Playout Policies for Monte-Carlo Go, Master's thesis, Institut für Kognitionswissenschaft, Universität Osnabrück (2010).
18. P. Baudiš, MCTS with Information Sharing, Master's thesis, Charles University in Prague - Faculty of Mathematics and Physics (2011).
19. B. Childs, J. Brodeur and L. Kocsis, Transpositions and Move Groups in Monte Carlo Tree Search, in *IEEE Symposium on Computational Intelligence and Games, 2008.* (Perth, AU, 2008), pp. 389–395.
20. P. Drake, The Last-Good-Reply Policy for Monte-Carlo Go, *International Computer Games Association Journal* **32**(4) (2009) 221–227.
21. R. Ramanujan, A. Sabharwal and B. Selman, On adversarial search spaces and sampling-based planning., in *ICAPS* **10**, (Toronto, CA, 2010), pp. 242–245.
22. M. Enzenberger, M. Muller, B. Arneson and R. Segal, FUEGO - an Open-Source Framework for Board Games and Go Engine Based on Monte-Carlo Tree Search, *Computational Intelligence and AI in Games, IEEE Transactions on* **2**(4) (2010) 259–270.
23. M. Harré, T. Bossomaier and A. Snyder, The Perceptual Cues that Reshape Expert Reasoning, *Scientific Reports* **2** (July 2012).